

STATE MIND


Valantis HOT

10-04-2024 - 23-04-2024



1. Project Brief		2
2. Finding Severity breakdown		3
3. Summary of findings		4
4. Conclusion		4
5. Findings report		5
Medium	Incorrect fee calculations	5
	HOT doesn't support the declared price range	6
	HOT does not handle rebasing scenarios.	7
	ALM doesn't work in AMM mode in the lack of oracles	8
	Swap in AMM mode becomes unavailable over time	9
	Complete or temporary shutdown of ammSwap	10
Informational	Unused errors in SOT contract.	11
	Redundant check	11
	minAMMFee limits configuration flexibility	11
	SOT.getReservesAtPrice() can revert in case of negative rebase	12
	Limit maxOracleUpdateDurationFeeds during deployment	13
	_checkSpotPriceRange insufficient check	13
	Unreachable code in integration with the current version of SovereignPool	14

1. Project Brief



Title	Description
Client	Valantis Labs
Project name	Valantis HOT
Timeline	10-04-2024 - 23-04-2024
Initial commit	8131ca3548aee6f28f46f52473e9390f5dce580c
Final commit	d80795d79706c58bfbf30b76c6f55d8b4c8cc579

Short Overview








Hybrid Order Type (HOT)* is a module in Valantis Sovereign Pools that allows external solvers/market makers to obtain signed quotes from the pool’s Liquidity Manager, which contain a trading volume, price, and AMM state updates like new spot price and fee parameters. These signed and time-limited HOT quotes can then be submitted on-chain by solvers to trade against the pool’s liquidity at the specified terms, with benefits such as deterministic liquidity for solvers, extra volume for liquidity providers, dynamic fees to protect against stale prices, and reduced exposure to pricing risks like top-of-block arbitrage.

Pools that enable HOT have two execution modes: a permissionless AMM and signed HOT quotes. Solvers obtain Signed Quotes from the pool’s Liquidity Manager via an API. These Quotes include a volume and price that the Solver can directly fill from the pool before expiry.

* – the project was previously called Solver Order Type (SOT), so in some findings we refer to previous name of contracts and components.

Project Scope

The audit covered the following files:

-  [SOT.sol](#)
-  [SOTOracle.sol](#)
-  [AlternatingNonceBitmap.sol](#)
-  [SOTConstants.sol](#)
-  [SOTParams.sol](#)
-  [TightPack.sol](#)
-  [SOTStructs.sol](#)

2. Finding Severity breakdown



All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	0 (0 fixed, 0 acknowledged)
Medium	6 (3 fixed, 3 acknowledged)
Informational	7 (4 fixed, 3 acknowledged)
Total	13 (7 fixed, 6 acknowledged)

4. Conclusion

During the audit of the codebase, 13 issues were found in total:

- 6 medium severity issues (3 fixed, 3 acknowledged)
- 7 informational severity issues (4 fixed, 3 acknowledged)

The final reviewed commit is d80795d79706c58bfbf30b76c6f55d8b4c8cc579

5. Findings report



MEDIUM-01

Incorrect fee calculations

Fixed at [be8312](#)

Description

Line: [SOT.sol#L749](#)

The current implementation of the contract calculates the fee according to the formula:

$$\frac{feeBipsToken = feeMinToken + feeGrowthInPipsToken * blockTimestamp - lastProcessedSignatureTimestamp}{100}$$

feeMinToken – specified in bips.

In general, percentage points represent the percentage difference between the two percentages.

Percent#1 = 10
Percent#2 = 11
pp difference = 11 - 10 = 1

So, **feeGrowthInPipsToken** – dimension is **pp / 100000** [SOT.sol#L118-L119](#).

Let's look at some examples:

feeMinToken = 0
feeGrowthInPipsToken = 0.1 pp
blockTimestamp = 10
lastProcessedSignatureTimestamp = 0

Calculation in classical pp
0 + 0.1pp * (10 - 0) = 0.1pp * 10 = 1 % = 100 bips

Calculation in contract's pp
0.1pp = 10000
0 + 10000 * (10 - 0) / 100 = 10000 * 10 / 100 = 1000 bips = 10%

Thus, the calculations don't correspond to the dimension of the basic values.

Impact: The contract manager may set too high a fee values or solvers may set an incorrect fee value because the dimension of the calculations is unclear.

Recommendation

We recommended dividing by 1000 instead of 100.

Client's comments

We have fixed it in the PR. feeGrowthE6 is meant to represent 1/100th of a BIPS. So 0.1% = 10 BIPS = 1000 E6 are the correct calculations

Description

Lines:

- [SOTParams.sol#L163-L164](#)
- [SOTParams.sol#L168](#)

The current pool declares a supported price range in [SOTConstants.sol#L17-L18](#). However, multiplication in **SOT.checkPriceDeviation()** function can lead to overflow, so the price has a limit **maxSqrtPrice = sqrt(2 ^ 256) / maxDeviationInBips = 2 ^ 128 / maxDeviationInBips**.

Impact: **SOT.setPriceBounds()**, **SOT.depositLiquidity()**, and **SOT._solverSwap()** functions revert when the **sqrtPrice** is more than **maxSqrtPrice**. So, core functionality may not be available.

Recommendation

We recommend calculating the deviation from the **sqrtPrice** and setting values to **maxOracleDeviationBips** based on **sqrtPrice**.

Description

HOT ALM declares support for rebasing tokens in the AMM swap mode.

Rebase tokens should be considered as tokens whose balance can increase or decrease at any time. Uniswap pools (and similar DEX pools) don't support rebase tokens: [UniswapV2 doc](#), [UniswapV3 doc](#).

HOT._ammSwap() doesn't calculate **_effectiveLiquidity** before starting the exchange, therefore the previous value of **_effectiveLiquidity** is based on outdated reserve balances. All liquidity studied in the invariant assumes a proportional division into all price ranges.

Let's look at a few cases of rebasing for HOT ALM.

Positive rebasing. **HOT** doesn't limit the **amountOut** and **SovereignPool** limits the **amountOut** on the amount of reserves.

There are cases when **SwapMath.computeSwapStep()** **amountOut** can use tokens from positive rebasing. The size of the rebase is not known in advance, therefore, the use of these tokens is incorrect because their accounting would change the invariant.

Negative rebasing. The previous value of **_effectiveLiquidity** is calculated based on large values of reserves. This leads to the calculation of the amount out via **SwapMath.computeSwapStep()** outputs more than necessary. Therefore, the real liquidity will end faster than the exchange reaches the limits of the range. This is an incorrect split of liquidity in the price range. LPs incur losses by swapping liquidity at the worst price.

SovereignPool allows 2 tokens with rebasing, therefore, more complex rebasing scenarios are possible when asset balances change in opposite directions.

The manager can pause the contract indefinitely. This blocks swaps and deposits of liquidity, however, after the resumption of the contract, the real balances of tokens may change significantly.

Recommendation

We recommend removing support for rebase tokens for AMM swap or adding support for various rebase cases.

Client's comments

Team:

We suggest extending the check in **HOT._ammSwap()** function

```
if (newLiquidity < _effectiveAMMLiquidity && pool.isRebaseTokenPool()) {
    _updateAMMLiquidity(newLiquidity);
}
```

We suggest skipping event emitting for **HOT.withdrawLiquidity()** in case **postWithdrawalLiquidity == preWithdrawalLiquidity**

```
if (postWithdrawalLiquidity < preWithdrawalLiquidity) {
    _updateAMMLiquidity(postWithdrawalLiquidity);
} else if (postWithdrawalLiquidity > preWithdrawalLiquidity) {
    emit PostWithdrawalLiquidityCapped(sqrtSpotPriceX96Cache, preWithdrawalLiquidity, postWithdrawalLiquidity);
}
```

Client:

Both are valid points, but in the interest of time we can skip these.

Adding `pool.isRebaseTokenPool()`, check inside to see if the statement could be equally or more costly than just doing a redundant update of the `effectiveAMMLiquidity` even when it is not required.

And skipping event emission for the equal case has negligible impact on the functioning of the protocol.

Partially fixed at: [b2549daae7aeaa2abac16e912256db709b9707cd](#)

Description

Lines:

- [SOT.sol#L497-L505](#)
- [SOT.sol#L578-L586](#)

According to the section "[Operating without the offchain components](#)" of the documentation, we understood that the **ammSwap** mode is available when an initial liquidity loading or there are no price feeds contracts. However, the current implementation doesn't allow adding liquidity to ALM if price feed contracts are not set. In addition, a **SOTOracle** contract cannot set the price feed after deployment. SOT should work without price feed, however, it is impossible to deploy a **SOTOracle** contract without price feed.

Impact: it is impossible to deploy a **SOT** contract and use it in the AMM mode if price feeds not exist

Recommendation

We recommend making **feedToken0** and **feedToken1** settable only once. If **feedToken0 == address(0)** or **feedToken1 == address(0)**, skip oracle price deviation checks in **SOT.depositLiquidity()** and **SOT.setPriceBounds()**. For greater reliability, the installation of price feeds can be performed only after obtaining permission from the manager and the liquidation provider.

Client's comments

Team:

Everything is fine, however, we recommend that you add a reset of **proposedFeeds** on behalf of LiquidityProvider. In this way, the manager will be able to offer other contract addresses and come to an agreement with LP.

Client:

Good suggestion but our codebase is now frozen and we also don't have the bytecode space to incorporate this

Partially Fixed at: [d80795d79706c58bfbf30b76c6f55d8b4c8cc579](#)

Description

Line: [SOT.sol#L750](#)

Casting to uint32 may overflow over time if **SOT** mode is not used or enabled. If the Valantis off-chain API was not deployed for the pool and at least 1 quote was not processed, then **SOT._getAMMFeelnBips()** return 0, because **feeMinToken{0,1} = 0**, **feeMaxToken{0,1} = 0**, **feeGrowthInPipsToken{0,1} = 0**. This is unexpected behavior because it is assumed that the AMM mod should work correctly without the off-chain API. If the Valantis off-chain API has been deleted, then an overflow will occur over time.

Let's take an example:

feelnBips dimentions has limit = 2^{32}

maxAMMFeeGrowthInPips has limit = 2^{16}

durationBeforeOverflow = $2^{32} / 2^{16} * 100 / 60 / 60 / 24 = 75.8518518519$ days

Recommendation

We recommend using **uint256** to calculate and cast after the maximum fee cap check and adding initialization of fee parameters during deployment.

```
function _getAMMFeelnBips(bool isZeroToOne) internal view returns (uint32 feelnBips) {
    SolverWriteSlot memory solverWriteSlotCache = solverWriteSlot;

    // Determine min, max and growth rate (in pips per second),
    // depending on the requested input token
    uint16 feeMin = isZeroToOne ? solverWriteSlotCache.feeMinToken0 : solverWriteSlotCache.feeMinToken1;
    uint16 feeMax = isZeroToOne ? solverWriteSlotCache.feeMaxToken0 : solverWriteSlotCache.feeMaxToken1;
    uint16 feeGrowthInPips = isZeroToOne
        ? solverWriteSlotCache.feeGrowthInPipsToken0
        : solverWriteSlotCache.feeGrowthInPipsToken1;

    // Calculate dynamic fee, linearly increasing over time
    uint256 feelnBipsTemp = uint256(feeMin) +
        Math.mulDiv(feeGrowthInPips, (block.timestamp - solverWriteSlotCache.lastProcessedSignatureTimestamp), 100);

    // Cap fee to maximum value, if necessary
    if (feelnBipsTemp > uint256(feeMax)) {
        feelnBipsTemp = uint256(feeMax);
    }
    feelnBips = uint32(feelnBipsTemp)
}
```

Description

Line: **SOT.sol#L779** – this uses the least available liquidity from reserves, cause in **_updateAMMLiquidity** we are setting **_effectiveAMMLiquidity** as the lowest one.

This can lead to the next 2 problems:

I. If there is no liquidity in one token, then **ammSwap** exchanges in the opposite direction are not possible. This could be as a result of a large exchange followed by withdrawal of the one remaining token, or as a result of the **LiquidityProvider** withdrawing all tokens from one reserve.

Example:

```
reserves0 = 100000000000000000000;  
reserves1 = 0;  
  
sot.swap(..., zeroForOne = false, amountIn = 1, ...);  
^-- will revert here, cause effectiveAMMLiquidity is zero
```

II. The rate of change in the **sqrtPriceSpot** price, as well as the corresponding ticks, changes not proportionally to the available liquidity, but only in proportion to the smallest of the liquidities. As a result, the fewer reserves in one of the tokens, the faster we move from one **sqrtPriceSpot** to the **sqrtPrice{High, Low}** boundary.

Example:

```
reserves0 = 50000000000000000000;  
reserves1 = 10000000000000000000000;  
  
(, uint256 amountOut) = sot.swap(..., zeroForOne = true, amountIn = 5.773502691896257884e18, ...);  
  
reserves0 = 10773502691896257884;  
reserves1 = 1479;  
  
// update effectiveAMMLiquidity and reduce reserves1  
sot.withdrawLiquidity(0, 700, address(this), 0, 0);  
  
// try to make arbitrage swap  
sot.swap(..., zeroForOne = false, amountIn = amountOut, ...);  
^-- will revert here with error message: SOT___ammSwap_invalidSpotPriceAfterSwap
```

Due to the existence of an upper and lower limit **sqrtPrice{High, Low}**, when conducting a large exchange, we limit the price of the attack. For example, liquidity will be very concentrated for a pair of stablecoins, which does not allow the price of one token to increase multiple. This condition makes an attack affordable.

POC: **SOTConcrete.t.sol.test_ammSwap_POC_swap_is_blocked_when_zero_liquidity()**,
SOTConcrete.t.sol.test_ammSwap_POC_amountOut_reverts_due_to_bound_error()

Recommendation

We recommend calculating the available liquidity for the two tokens separately.

An alternative solution is adding shares and sending them to **DEAD** addresses.

Client's comments

We think it is better not to implement the 10000 wei fix at the HOT level, so that we don't give future liquidity providers a false invariant that reserves can never go below 10000 wei. There is no way to prevent reserves from falling to 0 in case a negative rebase happens (since HOT supports rebase tokens). Therefore, all liquidity providers should have the necessary checks and security features mentioned above to prevent this bug.

We will highlight this issue explicitly in the documentation for liquidity providers, so that they can prepare their smart contracts accordingly. Moreover if a liquidity provider and signer wants to recover the security guarantees provided by this deposit, they can still do it at the periphery level by doing the following things –

1. Liquidity Provider deposits 10000 wei at the time of initialization, and doesn't implement the logic to be able to withdraw it.
2. Signer never signs a quote that makes the pool reserves fall below 10000 wei.

As a trusted signer's proper functioning is necessary for the HOT, this won't add any new trust assumptions. Therefore we don't need to enshrine the 10000 wei deposit logic into the HOT contracts.

INFORMATIONAL-01	Unused errors in SOT contract.	Fixed at 36f7f7
<div><div>Description</div><div>Lines:<ul style="list-style-type: none">• SOT.sol#L68• SOT.sol#L69• SOT.sol#L70Errors listed above are not used in SOT contract.</div><div>Recommendation</div><div>We recommend removing them.</div></div>		

INFORMATIONAL-02	Redundant check	Fixed at 17fd06
<div><div>Description</div><div>Line: SOT.sol#L226-L228 – the check doesn't work. If agrs.pool == address(0), it revert in SOT.sol#L218</div><div>Recommendation</div><div>We recommend correcting the existing check.</div></div>		

INFORMATIONAL-03	minAMMFee limits configuration flexibility	Acknowledged
<div><div>Description</div><div>Line: SOT.sol#L129 minAMMFee is a single minimum commission for token0, token1. However, pairs may have assets with different reliability. For such assets makes sense to set a minimum limit depending on tokenInput. For example, ETH/PEPE: If tokenInput = ETH (the pool receives a reliable asset) a fee is lower, if tokenInput = PEPE(the pool receives a less reliable asset) a fee is higher. In this way, the manager can take into account the risks of assets depending on swap direction.</div><div>Recommendation</div><div>We recommend separating the minAMMFee limits for each token.</div><div>Client's comments</div><div>This is a global minimum enforced for the minimum AMM fee across both tokens. To simplify the implementation and reduce bytecode, we prefer to state this intended behaviour in the docs.</div></div>		

Description

Line: [SOT.sol#L353](#)

Assume that **passiveReserve0** was zero before rebase:

```
(reserve0, reserve1) = ISovereignPool(pool).getReserves();

uint128 effectiveAMMLiquidityCache = _effectiveAMMLiquidity;

(uint256 activeReserve0, uint256 activeReserve1) = LiquidityAmounts.getAmountsForLiquidity(
    sqrtSpotPriceX96,
    sqrtPriceLowX96,
    sqrtPriceHighX96,
    effectiveAMMLiquidityCache
    ^-- Get active reserves with old _effectiveAMMLiquidity
);

uint256 passiveReserve0 = reserve0 - activeReserve0;
    ^--- Subtract old active reserves from new Reserves
uint256 passiveReserve1 = reserve1 - activeReserve1;
```

So if there is a negative rebase this function can revert because of underflow.

Same for **token1** Reserves.

Recommendation

We recommend adding proper **if-revert()** with a clear error message.

Client's comments

Team:

We suggest extending the check in **SOT.getReservesAtPrice()** function

```
uint128 calculatedLiquidity = _calculateAMMLiquidity();

if(calculatedLiquidity < effectiveAMMLiquidityCache && pool.isRebaseTokenPool()){
    effectiveAMMLiquidityCache = calculatedLiquidity;
}
```

Also, **_effectiveAMMLiquidity** at this function and in **SOT._ammSwap()** updates in case of negative rebase and doesn't update in other cases.

Client:

Adding pool.isRebaseTokenPool() check inside the if statement could be equally or more costly than just doing a redundant update of the effectiveAMMLiquidity even when it is not required.

Partially Fixed at: [6986cd43301cf9a49d36514a74e190b18a263978](#)

INFORMATIONAL-05	Limit <code>maxOracleUpdateDurationFeeds</code> during deployment	Acknowledged
------------------	---	--------------

Description

Lines:

- `SOTOracle.sol#L84-L85`

`maxOracleUpdateDurationFeed0` and `maxOracleUpdateDurationFeed1` are immutable state variables set during the deployment of the `SOT` in `SOTOracles.constructor()`. The constructor lacks value checks for those parameters.
 Impact: During deployment values of `maxOracleUpdateDurationFeed0` and `maxOracleUpdateDurationFeed1` can be zero, which will stop `SOT` from operating.

Recommendation

Firstly, the Chainlink heartbeat time ranges from 3600 to 86400 seconds (1-24 hours) thus, we recommend limiting the `maxOracleUpdateDurationFeed0` and `maxOracleUpdateDurationFeed1` to that range in `SOTOracle.constructor()`.
 Secondly, in cases of Chainlink heartbeat changing in the future, we recommend making the mentioned variables mutable and providing a setter function restricted to only `pool manager` role.

Client's comments

We want to keep the feeds as flexible as possible, in the future, some implementations might choose to use some other oracles than chainlink, and create wrappers around them so that they follow the `AggregatorV3Interface`.
 We also don't have the bytecode space to add mutable heartbeat variables, or setters.
 If a deployment has incorrect `maxOracleUpdateDuration` variables, it can be replaced by a new one after deployment.

INFORMATIONAL-06	<code>_checkSpotPriceRange</code> insufficient check	Fixed at 4c2312
------------------	--	---------------------------------

Description

Line:

`SOT.sol#L960` - we cannot set only one limit here (due to docs). So we should revert if one bound is set and another not. But it's not working if the upper bound is set and the lower is not, cause `MIN_SQRT_PRICE` will always greater than 0.
 Both bounds should be checked or both should be zero, otherwise revert should happen.

Recommendation

We suggest revising the verification code.

```

bool checkSqrtSpotPriceAbsDiff = _expectedSqrtSpotPriceUpperX96 != 0 || _expectedSqrtSpotPriceLowerX96 != 0;
bool isZero = _expectedSqrtSpotPriceUpperX96 == 0 || _expectedSqrtSpotPriceLowerX96 == 0;

if (checkSqrtSpotPriceAbsDiff && !isZero) {
    if (
        sqrtSpotPriceX96Cache > _expectedSqrtSpotPriceUpperX96
        || sqrtSpotPriceX96Cache < _expectedSqrtSpotPriceLowerX96
    ) {
        revert SOT___checkSpotPriceRange_invalidSqrtSpotPriceX96(sqrtSpotPriceX96Cache);
    }
} else if (checkSqrtSpotPriceAbsDiff && isZero) {
    revert SOT___checkSpotPriceRange_invalidBounds();
}

```

INFORMATIONAL-07	Unreachable code in integration with the current version of SovereignPool	Fixed at b25c1a
<div><div>Description</div><div>Lines: SOT.sol#L670 – SOT cannot be installed as an ALM module for the UniversalPool. SOT.sol#L680 – unreachable code, cause internalContext is always zero from SOT.getSwapFeeInBips().</div><div>Recommendation</div><div>We recommend removing these functions.</div></div>		

STATE MIND